ECE 730 Project Report: $M/G/k$ Queueing Simulation

Guanzhou Hu
guanzhou.hu@wisc.edu

University of Wisconsin – Madison

1 Introduction

Queueing is a ubiquitous problem in computer systems engineering. Many interesting challenges originate from the queueing problem, including but not limited to network packet transmission, request buffering, I/O device interface design, and load balancing [4]; it has also been heavily studied in the context of operations research [2]. Queueing models are an important application of probability theory and random processes. However, they are known to be mathematically hard to analyze; many performance metrics of (even simple) queueing models remain open problems [1, 6].

In this report, I study the general $M/G/k$ queue model. Specifically, I look into the 2-moment approximation formula for estimating the mean waiting time of customers in an $M/G/k$ queueing system, and implement a simulation framework to study the accuracy of this approximation.

Results show that the accuracy of the 2-moment approximation is strongly related to the *coefficient of variation* (CV, the ratio between standard deviation and mean) of the service time distribution. With a larger magnitude of CV, the approximation is less accurate and may deviate wildly from actual simulation results. Its accuracy is not sensitive to the mean of the service time distribution alone and the number of counters.

2 Queueing Models & Average Waiting Time

This section provides background on single queue models and their mean waiting time metrics.

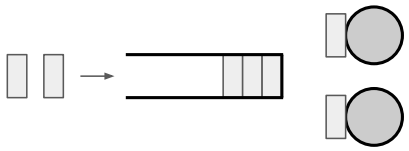

Figure 1: Demonstration of a queueing system.

2.1 Kendall's Notation

A queueing system is composed of three parts: customer arrival, queueing discipline, and service counters. *Customer arrival* depicts the arrival behavior of customers and is usually described as a random process. *Queueing discipline* controls which customers currently in the queue get served next; in this report, a first-come-first-serve (FIFO) discipline is assumed. Each *service counter* dequeues a customer when it is idle and spends a certain amount of

time, described by a service time distribution, to handle the customer. Figure 1 demonstrates such a queueing system.

Following Kendall's notation [4], we use three letters $A/S/k$ to describe a certain queueing model, where A represents the arrival process, S represents the service time distribution, and k is the number of counters. I assume a FIFO queueing discipline and an unbounded queue capacity. Also, note that when any counter is idle, a customer arriving at the system leaves the queue immediately and gets served at the counter with no queueing delay.

2.2 $M/M/k$ Queue & Average Waiting Time

An $M/M/k$ queue is one of the simplest queueing models. The first M stands for "Memoryless" or "Markovian" arrival, meaning that the arrival process is a memoryless Poisson process of rate λ. Each customer's arrival is independent of others, and the interval between any two arrivals is determined by an Exponential distribution of rate λ (i.e., of scale $\frac{1}{\lambda}$). The second M stands for "Memoryless" or "Markovian" departure, meaning that the service time of any customer at a counter is determined by an Exponential distribution of rate μ. The interplay between the two Exponential distributions on the two ends of an $M/M/k$ queue allows it to be described by a "birth-death" Markov chain whose states are the total number of customers in the system (including those in the queue and those at the counters) [7].

Let $k\mu$ denote the total rate of customer service at counters. An $M/M/k$ queue is non-convergent if $\lambda > k\mu$, i.e., the rate of customer arrival is larger than the aggregated rate of service available. In this case, queue length keeps growing over time. If $\lambda < k\mu$, the state of the system will reach a steady state distribution and hence various performance metrics can be analyzed.

The expected *waiting time* (or *delay*), W, denotes the average time that a customer spends waiting in the queue before getting served at a counter. W is a particularly interesting and useful metric because it helps determine the service-level agreement (SLA) of the queueing system and helps decide how many resources (i.e., counters) should a system create to reach a certain latency agreement. For an $M/M/k$ queue, W can be accurately computed. Define *server utilization* $\rho = \frac{\lambda}{k\mu}, \rho < 1$. The probability of an arriving customer being queued (i.e., seeing all counters occupied at the time of arrival) is given by Erlang's C-formula [7]:

$$C(k, \lambda, \mu) = \frac{\frac{(k\rho)^k}{k!} \cdot \frac{1}{1-\rho}}{\sum_{i=0}^{k-1} \frac{(k\rho)^i}{i!} + \frac{(k\rho)^k}{k!} \cdot \frac{1}{1-\rho}} = \frac{1}{1 + (1-\rho) \cdot \frac{k!}{(k\rho)^k} \cdot \sum_{i=0}^{k-1} \frac{(k\rho)^i}{i!}}. \tag{C}$$

The expected waiting time can then be calculated by:

$$\mathbb{E}[W^{M/M/k}] = \frac{C(k, \lambda, \mu)}{k\mu - \lambda}. \tag{$W^{M/M/k}$}$$

The $M/M/k$ queue does not provide sufficient flexibility and expressiveness in modeling many real systems. The limitation mainly comes from fixing the service time distribution as an Exponential distribution. In reality, the service time distribution might be determined by multiple complex factors, for example, the distribution of workload (job) sizes and the processing power of computational resources. Therefore, it is desirable to generalize the $M/M/k$ queue model by allowing arbitrary service time distributions.

2.3 $M/G/k$ Queue & 2-Moment Approximation

An $M/G/k$ queue is a queueing model where the arrival process is a memoryless Poisson process and the service time distribution is a general distribution G. It is assumed that $\lambda < \frac{1}{\mathbb{E}[G]}$ so that the model is convergent and allows steady-state analysis.

Closed-form solutions to many performance metrics of $M/G/k$ queues, including the expected waiting time, remain open problems as of today [6]. Various approximations have been proposed; the most classic approximation among them is the **2-moment approximation** by Lee and Longton [5]. It makes use of the $M/M/k$ counterpart to estimate the average waiting time of a given $M/G/k$ queue:

$$\mathbb{E}[W^{M/G/k}] \approx \frac{CV^2+1}{2} \cdot \mathbb{E}[W^{M/M/k}], \qquad (W^{M/G/k})$$

where CV is the *coefficient of variation* of G, i.e., $CV^2 = \frac{Var(G)}{(\mathbb{E}[G])^2}$, and $W^{M/M/k}$ is the waiting time of an $M/M/k$ with the same mean service time as G, i.e., $\frac{1}{\mu} = \mathbb{E}[G]$.

Previous work has proven that a closed-form solution cannot be obtained from only the first two moments of G [3]. In this report, I implement a simulation framework for the $M/G/k$ queueing model and study the accuracy of this approximation across different configurations.

3 Simulation with SimPy

I use SimPy, an event-driven simulation framework in Python, to simulate the behavior of $M/G/k$ queues and to empirically measure their average waiting time metrics.

3.1 SimPy Code Structure

My implementation contains ~300 lines of Python code. The complete code is provided in the Appendix §A. Besides SimPy, it depends on a few other packages: `numpy` and `scipy` for statistics-related support, `multiprocess` for parallelization, and `matplotlib` for visualization of results. The `MGkQueue` class is the core structure for the simulation and is mainly comprised of the following parts:

- A `simpy.Resource` resource object with capacity k that simulates a FIFO queue with k service counters.
- A `customer` helper function that represents the action taken by a single customer. The function first yields on requesting the above-mentioned resource; this corresponds to the time spent waiting in the queue. It then waits for a timeout sampled from a given distribution G before releasing its share of the resource; this corresponds to the time spent at a counter.
- An `arrival` loop that periodically triggers customer arrivals by putting a new `customer` entity into the simulation environment. It waits for a timeout sampled from an Exponential distribution of rate λ before entering the next iteration.

The code also takes care of parallelizing multiple trials of simulation on multiple CPU cores to speed up the overall simulation work.

3.2 Parameters Conversion for G

A particularly interesting problem I encountered during the implementation of the simulation infrastructure is the calculation of distribution parameters for G given a specified mean and standard deviation. Suppose that the set of Gs are distributions controlled by two input parameters. To fairly compare across different choices of G, I hold the mean and standard deviation of G constant a priori. To instantiate G with a `numpy` probability distribution class, I have to calculate the input parameters, e.g., the shape s and scale c for Weibull(s, c),

that produce the desired mean and standard deviation. This problem is generally known as *parameter estimation* for probability distributions.

For some distributions such as Lognormal, their parameters can be easily computed from mean and standard deviation. For others such as Weibull and TruncatedNormal, their parameters form non-linear relations with their moments; therefore, closed-form solutions might not be feasible. Instead, I use the `scipy` optimizers (e.g., `fsolve` and `root_scalar`) to numerically search for good-enough estimations of the parameters.

4 Simulation Results & Observations

Using the simulation infrastructure described in §3, I study three different choices of G: Lognormal(μ, σ), TruncatedNormal$(\mu, \sigma, 0, +\infty)$, and Weibull$(s, c)$. These distributions are common choices for modeling workload sizes, e.g., sizes of network packets and lengths of computation jobs.

I compare the simulation results of the average waiting delay against the 2-moment approximation, while varying three configurations: the mean service time $\mathbb{E}[G]$, the number of the counters k, and the coefficient of variation CV of G. For each configuration, I run 100 trials of simulation rounds with 500,000 arrivals each.

4.1 Varying CV^2

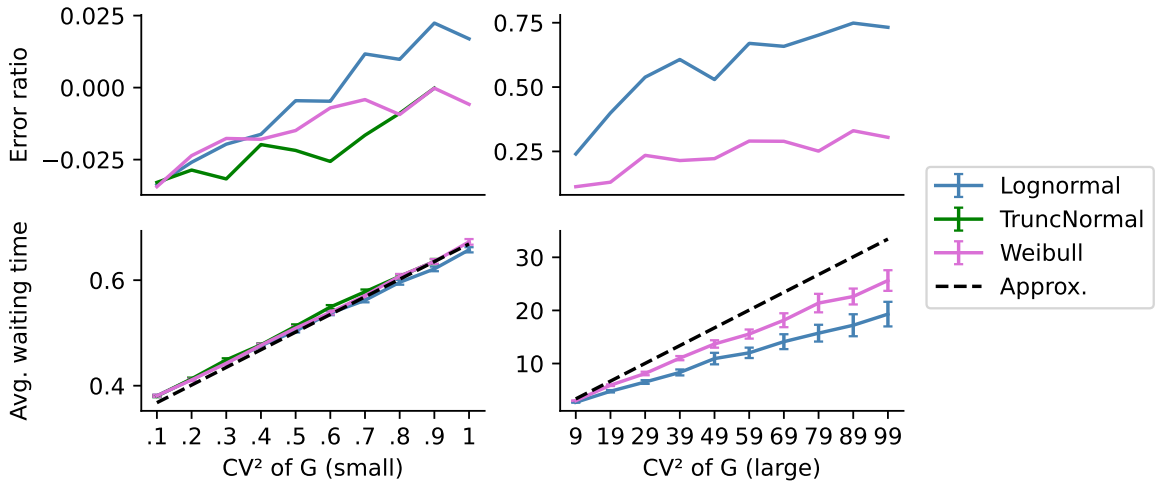

Figure 2: Simulation results, varying CV^2.

Figure 2 presents the results across different CV^2 scales. All distributions have $\mathbb{E}[G] = 1$ and there are $k = 10$ counters. The left half shows small ranges of CV^2 where the standard deviation is smaller than the mean, representing distributions that are considerably skewed towards the mean. The right half shows large ranges of CV^2 where the absolute value of standard deviation could be close to 10x the mean, representing distributions that span a wide range of service times. The bottom two figures show concrete values of average waiting time, as well as the 2-moment approximation by Formula $W^{M/G/k}$. The top two figures show relative error ratios, where a positive error ratio means overestimation and a negative error ratio means underestimation. Since TruncatedNormal distributions with large CV^2 are rare, we exclude them from the right half results.

The simulation results yield interesting observations. First, the approximation is accurate for small CV^2 values ≤ 1, yielding a maximum error ratio of ~3%. Second, the approximation

becomes less accurate and gives significant overestimation for larger CV^2 values > 1, with error ratios as high as 75%. This result matches earlier work [3]. Third, different distributions show different error ratios when CV^2 is large. The general trend is that the error ratio is worse for more "spread-out" distributions; to formalize this pattern, higher moments of the distributions might be required but are outside of the scope of this report.

4.2 Varying $\mathbb{E}[G]$ and k

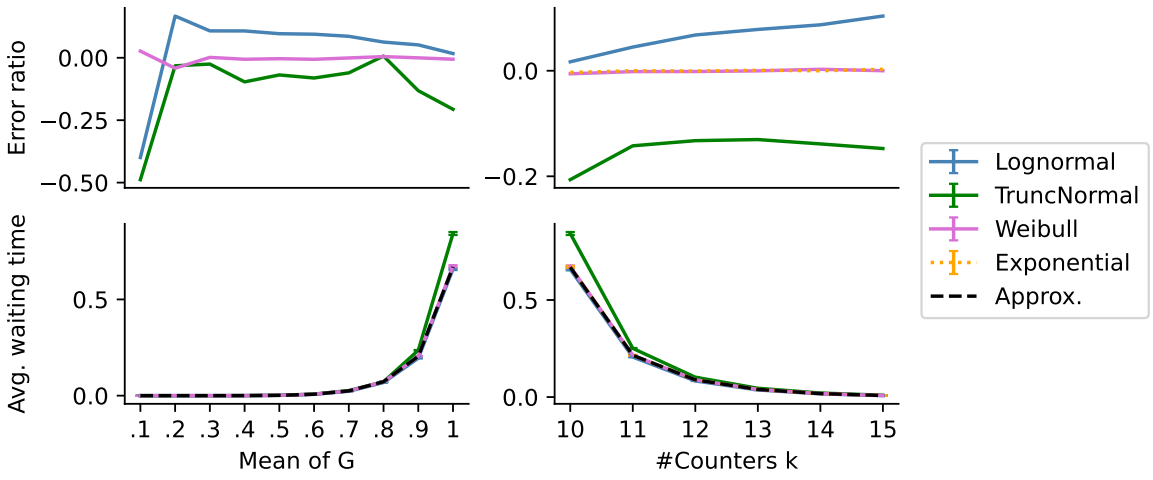

Figure 3: Simulation results, varying $\mathbb{E}[G]$ and k.

Figure 3 presents the results across different mean values and numbers of counters k. In the left half, all distributions have $CV = 1$ and there are $k = 10$ counters. In the right half, all distributions have $\mathbb{E}[G] = 1$ and $CV = 1$. The bottom two figures show concrete values while the top two figures show relative error ratios, in the same ways as in §4.1.

From these results, we can observe that the accuracy of the 2-moment approximation is mostly insensitive to both the mean of G and the number of counters k. For the Lognormal distribution, the error ratio increases with larger k. However, since the absolute values of waiting times are very small (< 0.01) for those k's, such inaccuracy is tolerable. For the TruncatedNormal distribution, the errors are mostly due to imperfections in parameter estimation. Note that the Exponential distribution is included in this experiment and it perfectly matches the approximation result, which is expected because the system degrades to an $M/M/k$ queue.

5 Conclusion

My simulation study shows that the 2-moment approximation for the average waiting time in $M/G/k$ queues is accurate for most of the configurations (despite the choice of the service time distribution G), as long as the absolute value of the coefficient of variation CV of G is relatively small (≤ 1). For larger absolute values of CV, however, it can produce a significant overestimation of up to 75%. The 2-moment approximation is insensitive to the mean value of G and the number of counters k.

The results indicate that the 2-moment approximation can be a useful estimation for the average waiting time in real-world queueing systems where the service times for requests are concentrated around a mean service time. In systems where the service times span a wide

range, the 2-moment approximation may produce significant overestimation; in these cases, more sophisticated methods will be needed.

References

[1] ARNOLD O. ALLEN. Chapter five - queueing theory. In ARNOLD O. ALLEN, editor, *Probability, Statistics, and Queueing Theory*, pages 149–233. Academic Press, 1978.

[2] Noah Gans, Ger Koole, and Avishai Mandelbaum. Telephone call centers: Tutorial, review, and research prospects. *Manufacturing & Service Operations Management*, 5(2):79–141, 2003.

[3] Varun Gupta, Mor Harchol-Balter, Jim Dai, and Bert Zwart. On the inapproximability of m/g/k: Why two moments of job size distribution are not enough. *Queueing Systems*, 64:5–48, 08 2010.

[4] David G. Kendall. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 24(3):338 – 354, 1953.

[5] A. M. Lee and P. A. Longton. Queueing processes associated with airline passenger check-in. *Journal of the Operational Research Society*, 10(1):56–71, 1959.

[6] Wikipedia contributors. M/g/k queue — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=M/G/k_queue&oldid=1187803549, 2023. [Online; accessed 11-December-2023].

[7] Wikipedia contributors. M/m/c queue — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=M/M/c_queue&oldid=1170369278, 2023. [Online; accessed 11-December-2023].

A Complete Python Code

```
import argparse
import math
import statistics
import pickle
from multiprocess import Pool
import simpy
import numpy as np
from scipy import stats
from scipy.special import gammaln, logsumexp
from scipy.optimize import root_scalar, fsolve
import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt


class Distribution(object):
    def __init__(self):
        self.disp = "Base"
        self.mean = -1
        self.stdev = -1
```

```
    def name(self):
        return "base"

    def sample(self, rng):
        raise RuntimeError("calling .sample() on base class")


class DistExponential(Distribution):
    def __init__(self, rate):
        super().__init__()
        self.disp = "Exponential"
        self.scale = 1.0 / rate
        self.mean = self.scale
        self.stdev = self.scale
        self.cv2 = 1.0

    def name(self):
        return "exponential"

    def sample(self, rng):
        return rng.exponential(scale=self.scale)


class DistLognormal(Distribution):
    def __init__(self, target_m, target_cv2):
        super().__init__()
        self.disp = "Lognormal"
        self.mean = target_m
        self.stdev = target_m * math.sqrt(target_cv2)
        self.cv2 = target_cv2
        self.normal_si = math.sqrt(math.log(target_cv2 + 1))
        self.normal_mu = math.log(target_m) - (self.normal_si**2 / 2.0)
        # print(
        #     self.name(),
        #     (self.mean, self.stdev**2),
        #     stats.lognorm(self.normal_si, scale=math.exp(self.normal_mu)).stats(),
        # )

    def name(self):
        return f"lognormal-{self.mean:.1f}-{self.cv2:.1f}"

    def sample(self, rng):
        return rng.lognormal(mean=self.normal_mu, sigma=self.normal_si)


class DistTruncNormal(Distribution):
    @staticmethod
    def conversion(mean, stdev):
        """
        Numerically look for parameters given mean and stdev.
        Ref: https://stats.stackexchange.com/questions/408171
        """
        p_phi = lambda z: (1 / math.sqrt(2 * math.pi)) * math.exp(-0.5 * z**2)
        c_phi = lambda z: 0.5 * (1 + math.erf(z / math.sqrt(2)))
```

```
    def eqs(p):
        m, s = p
        r = m / s
        eq1 = m + s * (p_phi(r) / (1 - c_phi(-r))) - mean
        eq2 = (
            s**2
            * (
                1
                - ((m * p_phi(r) / s) / (1 - c_phi(-r)))
                - (p_phi(r) / (1 - c_phi(-r))) ** 2
            )
            - stdev**2
        )
        return (eq1, eq2)

    x, _, ier, mesg = fsolve(
        eqs, (mean, stdev), maxfev=2000, xtol=1e-16, full_output=True
    )
    # if ier != 1:
    #     print(mesg)
    m, s = x

    return m, s

def __init__(self, target_m, target_cv2):
    super().__init__()
    self.disp = "TruncNormal"
    self.mean = target_m
    self.stdev = target_m * math.sqrt(target_cv2)
    self.cv2 = target_cv2
    self.normal_mu, self.normal_si = DistTruncNormal.conversion(
        self.mean, self.stdev
    )
    r = self.normal_mu / self.normal_si
    self.dist = stats.truncnorm(
        -r, np.inf, loc=self.normal_mu, scale=self.normal_si
    )
    # print(
    #     self.name(),
    #     (self.mean, self.stdev**2),
    #     (self.normal_mu, self.normal_si),
    #     self.dist.stats(),
    # )

def name(self):
    return f"truncnormal-{self.mean:.1f}-{self.cv2:.1f}"

def sample(self, rng):
    if self.normal_mu < -5.0:  # while-loop might be prohibitively slow
        return self.dist.rvs(random_state=rng)
    else:
        while True:
            t = rng.normal(loc=self.normal_mu, scale=self.normal_si)
            if t >= 0:
                return t
```

```
class DistWeibull(Distribution):
    @staticmethod
    def conversion(mean, stdev):
        """
        Numerically look for parameters given mean and stdev.
        Ref: https://github.com/scipy/scipy/issues/12134
        """
        log_mean, log_std = np.log(mean), np.log(stdev)

        def r(c):
            logratio = (  # np.pi*1j is the log of -1
                logsumexp([gammaln(1 + 2 / c) - 2 * gammaln(1 + 1 / c), np.pi * 1j])
                - 2 * log_std
                + 2 * log_mean
            )
            return np.real(logratio)

        res = root_scalar(
            r, method="bisect", bracket=[1e-300, 1e300], maxiter=2000, xtol=1e-16
        )
        assert res.converged
        c = res.root
        scale = np.exp(log_mean - gammaln(1 + 1 / c))
        return c, scale

    def __init__(self, target_m, target_cv2):
        super().__init__()
        self.disp = "Weibull"
        self.mean = target_m
        self.stdev = target_m * math.sqrt(target_cv2)
        self.cv2 = target_cv2
        self.alpha, self.scale = DistWeibull.conversion(self.mean, self.stdev)
        # print(
        #     self.name(),
        #     (self.mean, self.stdev**2),
        #     stats.weibull_min(self.alpha, scale=self.scale).stats(),
        # )

    def name(self):
        return f"weibull-{self.mean:.1f}-{self.cv2:.1f}"

    def sample(self, rng):
        return self.scale * rng.weibull(self.alpha)


ARR_RATE = 9.0
ARR_DIST = DistExponential(ARR_RATE)

CV2_SRANGE = [0.1 * i for i in range(1, 11)]
CV2_LRANGE = [9 + 10 * i for i in range(10)]
MEAN_RANGE = [0.1 * i for i in range(1, 11)]
K_RANGE = [i for i in range(10, 16)]
G_K_LIST = (
    [(DistExponential(1.0), 10)]
    + [(DistLognormal(1.0, cv2), 10) for cv2 in CV2_SRANGE]
```

```
    + [(DistLognormal(1.0, cv2), 10) for cv2 in CV2_LRANGE]
    + [(DistLognormal(mean, 1.0), 10) for mean in MEAN_RANGE]
    + [(DistTruncNormal(1.0, cv2), 10) for cv2 in CV2_SRANGE]
    + [(DistTruncNormal(mean, 1.0), 10) for mean in MEAN_RANGE]
    + [(DistWeibull(1.0, cv2), 10) for cv2 in CV2_SRANGE]
    + [(DistWeibull(1.0, cv2), 10) for cv2 in CV2_LRANGE]
    + [(DistWeibull(mean, 1.0), 10) for mean in MEAN_RANGE]
    + [(DistExponential(1.0), k) for k in K_RANGE]
    + [(DistLognormal(1.0, 1.0), k) for k in K_RANGE]
    + [(DistTruncNormal(1.0, 1.0), k) for k in K_RANGE]
    + [(DistWeibull(1.0, 1.0), k) for k in K_RANGE]
)

NUM_ARRIVALS = 500000
NUM_TRIALS = 100


class MGkQueue(object):
    def __init__(self, rng, env, a, g, k, logging=False):
        self.rng = rng
        self.env = env
        self.system = simpy.Resource(env, capacity=k)
        self.cus_map = dict()
        self.next_id = 0
        self.a = a
        self.g = g
        self.results = []
        self.logging = logging

    def print_stats(self, extra=""):
        print(
            f" [{len(self.system.queue)}] "
            + f"{self.system.count:>2d}/{self.system.capacity:<2d} {extra}"
        )

    def customer(self, cus_id, req):
        # wait in queue if no counters available
        yield req
        start = self.cus_map[cus_id]
        del self.cus_map[cus_id]
        elapsed = self.env.now - start
        if self.logging:
            self.print_stats(extra=f"waited {elapsed:.3f}")
        self.results.append(elapsed)
        # being serviced at a counter
        yield self.env.timeout(self.g.sample(self.rng))
        self.system.release(req)

    def arrival(self):
        while True:
            if self.logging:
                self.print_stats(extra="arrive")
            # put a new customer in the form of a request on the resource
            self.cus_map[self.next_id] = self.env.now
            req = self.system.request()
            self.env.process(self.customer(self.next_id, req))
```

```
            # Poisson process: randomly wait for an exponential interval
            self.next_id += 1
            if self.next_id == NUM_ARRIVALS:
                break
            yield self.env.timeout(self.a.sample(self.rng))


def simulate(a, g, k, logging):
    rng = np.random.default_rng()
    env = simpy.Environment()
    queue = MGkQueue(rng, env, a, g, k, logging)
    env.process(queue.arrival())
    env.run()
    return sum(queue.results) / len(queue.results)


def do_simulations(logging):
    for g, k in G_K_LIST:
        print(f"g={g.name()} k={k}")
        with Pool() as pool:
            ws = pool.map(
                lambda _: simulate(ARR_DIST, g, k, logging),
                range(NUM_TRIALS),
            )
        with open(f"{g.name()}-{k}.pkl", "wb") as fpkl:
            pickle.dump(ws, fpkl)


def read_results():
    results = dict()
    for g, k in G_K_LIST:
        results[(g, k)] = dict()
        with open(f"{g.name()}-{k}.pkl", "rb") as fpkl:
            l = pickle.load(fpkl)
            # l.sort()
            # throw = int(len(l) * 0.05)  # remove 10% outliers
            # l = l[throw:-throw]
            avg = sum(l) / len(l)
            cil = 1.96 * statistics.stdev(l) / math.sqrt(len(l))  # 95% CI
            results[(g, k)] = (avg, cil)
    return results


def approximation(g, k):
    dep_rate = 1.0 / g.mean
    util = ARR_RATE / (k * dep_rate)
    denom_sum = sum(((k * util) ** i / math.factorial(i) for i in range(k)))
    prob = 1.0 / (1 + (1 - util) * (math.factorial(k) / ((k * util) ** k)) * denom_sum)
    w_MMk = prob / (k * dep_rate - ARR_RATE)
    cv = g.stdev / g.mean
    w_MGk = ((cv**2 + 1) / 2.0) * w_MMk
    return w_MGk


def plot_results(results):
    for g, k in G_K_LIST:
```

```
            approx = approximation(g, k)
            avg_sim, cil_sim = results[(g, k)]
            print(
                f"g={g.name()} k={k}  approx: {approx:.3f}  "
                + f"sim: {avg_sim:.3f} ±{cil_sim:.3f}"
            )


    if __name__ == "__main__":
        parser = argparse.ArgumentParser()
        parser.add_argument(
            "-l",
            "--logging",
            action="store_true",
            help="if set, print log during simulation",
        )
        parser.add_argument(
            "-p",
            "--plot",
            action="store_true",
            help="if set, do the plotting phase",
        )
        args = parser.parse_args()

        if not args.plot:
            do_simulations(args.logging)
        else:
            results = read_results()
            plot_results(results)
```